Hindawi
Security and Communication Networks
Volume 2022, Article ID 2031924, 15 pages
https://doi.org/10.1155/2022/2031924

Research Article

GAXSS: Effective Payload Generation Method to Detect XSS Vulnerabilities Based on Genetic Algorithm

Zhonglin Liu, Yong Fang, Cheng Huang, and Yijia Xu

School of Cyber Science and Engineering, Sichuan University, Chengdu, Sichuan, China

Correspondence should be addressed to Cheng Huang; opcodesec@gmail.com

Received 9 October 2021; Revised 24 February 2022; Accepted 28 February 2022; Published 30 March 2022

Academic Editor: Helena Rifà-Pous

In the fields of social networking, media, and management, web applications on the Internet play a very indispensable role. A large amount of personal privacy information and login tokens make web applications often targeted by hackers. Cross-site scripting attacks are the most common method used to steal data from web applications. To solve the security risks caused by cross-site scripting vulnerabilities, security personnel need to actively discover these vulnerabilities to better defend against the harm. We proposed a novel genetic algorithm-based fuzzing scheme to address this problem. First, a small number of initial attack vectors are generated according to the interactive environment of the web application and then the attack vectors are sequenced into genes. Combined with the grammatical structure features of cross-site scripting and common bypass methods, the gene sequences are iteratively optimized and improved. Finally, the generated high-quality vectors are used to detect potential cross-site scripting threats in the application (we named the implementation of this approach GAXSS). The method we proposed can automatically detect the vulnerability of page interaction points and can obtain better detection results without a large number of test dictionaries, and the time cost is also reasonable. We have conducted vulnerability tests on many common open-source web applications, with a precision rate of 1.0 and an accuracy rate over 0.98. In addition, we also compared GAXSS with other well-known scanners and state-of-the-art detection methods. Its comprehensive performance is better, and it can effectively detect vulnerabilities.

1. Introduction

Internet technology has played an irreplaceable and important role in people's production and life. According to the 47th "Statistical Report on China's Internet Development Status" issued by the China Internet Network Information Center (CNNIC) in 2021 [1], as of December 2020, the number of Internet users in China reached 989 million and the Internet penetration rate reached 70.4.

The widespread use of web technologies brings with it security risks and vulnerabilities. An OWSAP survey showed that [2] cross-site scripting (abbreviated as XSS) attacks have always been among the top three web security vulnerabilities in the top 10 web security threats. Therefore, the detection and protection of XSS vulnerabilities are an important research element in web security. Attackers generally exploit XSS vulnerabilities to launch attacks on the victim's browser by controlling the browser to execute malicious JavaScript to achieve information theft, unauthorized access, RCE (Remote Command Execution), and other intents. In recent years, incidents such as the Renren worm [3], Samy worm [4], Boonana worm [5], and Spaceflash worm [6] were caused by XSS vulnerabilities in the system, leading to the successful implementation of worm attacks by attackers, resulting in massive information leakage and serious financial losses.

Researchers have proposed a variety of detection methods to detect vulnerabilities, with fuzz testing being one of the most common and effective methods. Fuzzing technology was first proposed by Miller in 1989 [7]. It is used to test the reliability of Unix systems and to detect vulnerabilities in protocols and applications. Fuzzing technology mainly uses a large number of unexpected randomized vectors to make the system produce

unconventional responses and then judge whether the system is vulnerable according to the corresponding results. In addition, the fuzzing technology does not require any prior knowledge about the target software system and has the characteristics of strong detection ability for unknown vulnerabilities and a high degree of automation [8].

XSS attack types generally contain persistent attacks (stored XSS), nonpersistent attacks (reflected XSS), and DOM-based XSS attacks [9], and the attack payload can also bypass some of the server's security detections by encoding, morphing, and replacing keywords [10]. We consider the diversity of XSS vulnerability attack patterns, and, to improve the accuracy of fuzzing tests, it is necessary to consider how to generate valid test vectors from the attacker's perspective. We need to consider the construction of a syntactic attack payload and the mutation of the attack payload to bypass some of the security filtering mechanisms present in the system. Many researchers try to use log analysis, machine learning, semantic analysis, and other methods to fuzz test vulnerabilities, hoping to improve the performance in certain aspects [11–13]. In this paper, the genetic algorithm is introduced as a test vector generation method in XSS vulnerability detection, which improves the vulnerability trigger rate and enhances the depth of fuzz testing, so that the model can exert excellent performance in the detection of XSS vulnerabilities in web applications.

In this paper, we propose a novel method for generating fuzzy test vectors based on genetic algorithms, called GAXSS. Our main contributions are as follows:

(i) We have proposed a method based on taint tracking, which generates different types of test data for the test points of different HTML pages injecting positions, so as to reduce the low-quality parent set in the genetic algorithm process and improve the computational efficiency.

(ii) Through the research of XSS vulnerability mining technology, we have integrated the modification of bypassing the security detection method into the mutation step of the genetic algorithm. This method can greatly increase the probability that the attack vector bypasses the security strategy and the payload triggers the vulnerability.

(iii) In order to ensure that the excellent attack vector structure is preserved, we improved in each round of iteration, and all the generated offspring individuals are sorted together with the parent individuals. In this way, individuals with high fitness are selected as new parents to ensure that relatively high-quality individuals will not be covered by offspring individuals.

(iv) We designed an advanced detection method, which is based on genetic algorithm and XSS structure analysis, through multiple evolutions and mutations of the initial payload set to dynamically generate more effective payloads and use it to detect vulnerabilities.

The rest of this paper is organized as follows. In Section 2, the related work of this research direction is introduced. In Section 3, the principle of XSS vulnerability and the targeted sample generation and mutation scheme are introduced. In Section 4, we will describe in detail the test vector generation scheme based on the genetic algorithm. The related experiments and evaluations of this method will be shown in Section 5. Section 6 is the summary of our work and discusses the future work direction.

2. Related Work

Many researchers have proposed many targeted detection methods for XSS vulnerabilities, hoping to use efficient and accurate detection methods to find vulnerabilities, to minimize the impact of XSS vulnerabilities on network security. Taking into account the different application scenarios of XSS detection methods, we mainly classify the current research work into active defense and passive defense.

2.1. Active Defense. "Active defense" means a program that can actively discover the vulnerabilities of the software before the application is publicly released. H. Shahriar et al. proposed MUTEC, which uses mutated test cases to test the PHP source code to detect whether the filter can be bypassed [14]. The Pixy method proposed by N. Jovanovic et al. analyzes traffic, process components, and context-sensitive data streams to find vulnerabilities in PHP applications [15]. F. Duchene et al. used model reasoning and evolutionary fuzzy testing to detect XSS vulnerabilities [16]. The model obtains application behavior knowledge in reasoning and uses a genetic algorithm to generate fuzzy test input. Z. Tang et al. proposed a method based on lexical mutation. The engine fuzzes the seed to generate fuzzing input for XSS testing [17]. Y. Zhang et al. created an XSS attack vector library to detect XSS vulnerabilities introduced by HTML5 into web applications [18]. Xin et al. proposed a dynamic detection technology, which uses grammatical structure features to simulate attacks on the target and finally detects whether the attack is successful or not [19]. Many of the above-mentioned methods of researchers can effectively detect XSS vulnerabilities, but the effect of the method is basically proportional to the size of the attack vector, and it will take more time to use a large vector library.

2.2. Passive Defense. The discovery mechanism of "Passive defense" is to detect incoming XSS attacks in real time after the application is online. Because this kind of scheme needs to produce certain interventions to the web application program, it will have a certain influence on the robustness and stability of the application program itself. The Noxes [20] proposed by E. Kirda et al. and the schemes proposed by A. Barth [21] all act on the client, and most of the research work is focused on "cross-site request forgery (CSRF)." In addition, solutions such as XSSDS [22] and XSS-GUARD [23] all use Firefox browsing components to accurately identify the injected scripts in the web page. G. Wassermann et al. performed string analysis on the tainted information

flow, and they also detected whether the untrusted part of the code was called the JavaScript interpreter [24]. Zhong et al. also proposed converting the payload submitted by the user into a graph to detect whether there is a threat of cross-site attack [25]. These methods are based on browser components, browser open-source code, or passively waiting to be attacked before detection and are relatively limited in terms of security defense.

2.3. Test Case Generation. The generation of test cases is often regarded as the standard of code coverage. In the research of web application fuzzing, R. Hammersland et al. proposed a semiautomatic web application fuzzing scheme [26], which requires a certain labor cost to analyze logs and responses. J. Bozic et al. proposed a combination scheme of attack grammar mode restricted by constraints, which can better improve the attack coverage of web security testing [27]. F. Duchene et al. proposed using an improved fuzzing method based on fuzzy reasoning when discovering XSS vulnerabilities [28], but this method can only be effective for reflective XSS vulnerabilities and is still in the process of further research. Muhammad et al. used vulnerability features to generate various feature vectors to automatically scan black-box web vulnerabilities and can also effectively find many web vulnerabilities [29]. Wang et al. used taint tracking technology to effectively detect Dom-type XSS vulnerabilities based on dynamic methods and demonstrated the effectiveness of their model in experiments [30].

Through the analysis of these studies, we found that the method of generating and simulating attacks using attack vectors can actively discover XSS vulnerabilities in the application more effectively. At the same time, the detection schemes proposed by C. Chen et al. and F. Duchene et al. both introduced genetic algorithms to effectively improve the efficiency and accuracy of fuzzing test sample generation [28, 31]. Therefore, in this paper, we proposed a fuzzing sample generation method based on an improved genetic algorithm for the detection of XSS vulnerabilities. It is hoped that attack vectors can be effectively found through nonlarge vector libraries, thus actively discovering vulnerabilities.

3. Background and Motivation

3.1. XSS Principle. The essence of XSS vulnerabilities is the injection of HTML and JavaScript code. The attacker inserts malicious code into the web page by submitting a form or making a request. When the victim browses the web page, the malicious code embedded in the page will be executed, so as to achieve the purpose of the XSS attack [32].

Through our analysis, we found that XSS vulnerabilities are usually caused by developers failing to perform compliance verification and effective filtering on user input or URL submission, allowing the page to directly insert the requested content into the DOM tree temporarily or continuously. This leads to malicious scripts being loaded and executed, resulting in a vulnerability. For example, if an attacker introduces < *img%20src = 1%20onerror = alert(1*a) > in the parameters of a GET/POST request, it may cause a pop-up window on the web page to successfully exploit the vulnerability and execute the attacker's malicious code.

According to the way the attacker injects malicious scripts into the web application, we have divided the types of XSS into continuous attacks (stored XSS), nonpersistent attacks (reflective XSS), and DOM-based XSS attacks. The three types of XSS are used in different ways [9].

(i) *Continuous Attacks*. They are also known as stored XSS. This type of attack utilizes the server's feature that the content entered by the user is stored in the database, such as forum replies, messages in the message board, and remarks in account registration. If the content in these databases is not strictly processed, the page directly displays the data in the database on the page, which will cause the malicious script in the content to be executed.

(ii) *Nonpersistent Attacks*. They are also known as reflective XSS. The reflective XSS attack process generally involves the attacker constructing an attack URL in advance and then tricking the victim to visit the URL, and the URL already contains malicious scripts that will be executed when the victim opens it. For instance, in a search box, the page will display the keywords that need to be searched. If the web application does not effectively filter the keywords searched by the user, there may be vulnerabilities.

(iii) *DOM-Based XSS Attacks*. They are a vulnerability based on the Document Object Model (DOM). There are many objects in the DOM, some of which can be manipulated by the user, such as URI and location. The client-side script program can dynamically check and modify the content of the page through the DOM. It does not rely on submitting data to the server. The client can execute the data obtained in the DOM locally. The use of DOM-based XSS is similar to the use of reflective XSS, but the difference is that the content it submits is for interactive requests locally in the browser, without communicating with the server.

3.2. Payload Component. The number of XSS attack vectors is large, but they are not arbitrarily generated. Attack vectors have certain semantic rules and characteristics. Through our collection and study, we found that the attack vectors mainly behave as follows:

(i) The core of the XSS attack vector is to contain the attack code. When the attack vector is successfully output to the browser, these attack scripts can be loaded and executed normally.

(ii) XSS attack vectors can execute attack codes for different purposes.

(iii) The structure of the attack vector needs to conform to the context of the output point, so that the browser can parse the script code normally.

All in all, the components of the XSS attack vector contain at least the attack code, and the attack vector will also contain HTML tags, attributes, or events in most cases. In some cases, the attack vector also needs to contain closed characters in order to comply with the output context. For our convenience to describe the composition of the XSS attack vector, this paper summarized the components of the attack vector shown in Table 1. We used the definition in Table 1 to split an example of the XSS attack vector parts as shown in Figure 1:

(a) Tags: in HTML, special attributes are attached to HTML tags, and all tags that can contain special attributes can become elements of XSS attack vectors. The tags often used by attackers are shown in Table 2.

(b) Special attributes: attackers can use special attributes to trigger the browser's JavaScript parser, analyze and execute the JavaScript attack code in the attack vector, and realize XSS attacks. The special attributes commonly used in HTML are shown in Table 3.

(c) Pseudoprotocols: in HTML, special attributes require a pseudoprotocol to load the attack code. The pseudoprotocols commonly used in attack vectors are shown in Table 4. The usage of the JavaScript pseudoprotocol is shown in Figure 1 as an example. When the data pseudoprotocol transmits data, the content needs to be encrypted with Base64, so the attack code format based on the data pseudoprotocol is *data:text/html;base64,[malicious code]*. The code in [malicious code] is the code after Base64 encryption.

(d) Malicious code: malicious code is the core component of the attack vector. It is composed of JavaScript code, and the code will be different for different attack purposes. In this paper, it is mainly used for vulnerability detection in web applications, so the pop-up and jump JavaScript functions are selected as the component factors of the attack vector as shown in Table 5.

(e) Closed characters: the closed character is a significant part of the XSS attack vector. By closing tags, attributes, or other codes in the original HTML and changing the DOM structure, so that the injected attack vector conforms to the context of the output location, it is possible to successfully execute the attack code in the attack vector.

(f) Events: event is an event-driven attribute of tags in HTML, and JavaScript code can be executed by driving a specific event. Event-driven and executed code does not require pseudoprotocols or script tags. For instance, the *onerror* event is a common event that can directly trigger execution, while events such as *onmouseover* need to be triggered after the mouse passes over the component. Common events are shown in Table 6.

Table 1: XSS components.

ID	Components
1	Tags
2	Special attributes
3	Pseudoprotocols
4	Malicious code
5	Closed characters
6	Events

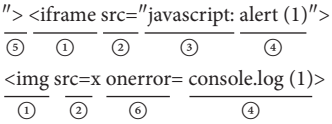

Figure 1: Attack vector split example (the labels correspond to Table 1).

3.3. Injection Location. The key to the successful execution of cross-site scripting is that the structure of the attack vector conforms to the output context. In this paper, we hope to use a genetic algorithm to generate an effective attack payload. Therefore, when generating the initial population, the generation of invalid individuals is minimized as much as possible, thereby improving the algorithm's convergence rate. Through analysis, we divided the cross-site scripting injection position into three categories: between script tag, in the HTML tag, and outside the tag.

Each time we generate the initial population, we will use the taint tracking technology to detect the relative position of the injection point and then generate the corresponding initial test population according to different types of positions, which can greatly reduce the initial state from being disturbed by useless data and algorithms can find high adaptive solutions quickly:

(i) **Between script tags**: when the injection point is between script tags, only the last sentence of the input point needs to be closed, and the attack script can be directly constructed without guiding the browser to parse the JavaScript code through the events or attributes of the HTML tag. Finally, we need to close the sentence after the input point, so that the syntax of this part of the JavaScript code is legal and will not report errors (JavaScript code is different from HTML code; once the JavaScript code reports a syntax error, it will not run). The attack vector form under this type of output point is as follows: *";eval(alert(1));//*.

(ii) **In the HTML tags**: when the injection point is in the HTML tags, there are two situations. One situation is to close the current label and embed the attack script in the new label, for example, *'><img src = 1 onerror = console.log(1)>*. Another situation is to close the attributes or events of the current label and embed attack scripts in the new attributes or events, for example, *"onload = alert(1)>*. In both cases, there is no need to consider the issue of backward closure, because the browser can

Table 2: Component: tags.

Component	Contents
Tags	1.<*a*>, 2.<*p*>, 3.<img>, 4.<body>, 5.<script>, 6.<var>, 7. , 8.<object>, 9.<input>, 10.<select>, 11.<iframe>, 12.<frameset>, 13.<embed>, 14.<svg>, 15.<video>, 16.<audio>

Table 3: Component: special attributes.

Component	Contents
Special attributes	1.src, 2.dynsrc, 3.lowsrc, 4.href, 5.action, 6.data, 7.background, 8.formaction, 9.poster, 10.code, 11.location, 12.name

Table 4: Component: pseudoprotocols.

Component	Contents
Pseudoprotocols	1.Javascript, 2.data

Table 5: Component: malicious code.

Component	Contents
Malicious code	1.alert(), 2.confirm(), 3.prompt(), 4.self.location, 5.top.location, 6.location.href

Table 6: Component: events.

Component	Contents
Events	1.onerror, 2.onclick, 3.onblur, 4.onmousedown, 5.onmouseup, 6.onmouseover, 7.onmousemove

automatically complete tag completion and closure during the process of parsing HTML code. Even if it is not closed, it will not affect the execution of the exploit code in the attack vector.

(iii) **Outside the tag**: when the injection point is outside the tag, such as between <body></body>, we do not need to consider the code closure before and after and directly introduce the attack script through the new tag, such as <*a href = ' javascript: alert(1)'*>.

3.4. Attack Vector Mutation. Many developers have taken into account the legality of user-submitted content and filtered illegal content when developing web applications, but there are still some ways to bypass these security policies and launch XSS attacks on web applications. Through the research of a large number of web filtering mechanisms and attack vector mutation methods, in this paper, we summarized the four commonly used and effective mutation methods shown in Table 7 for use in the mutation process part of the genetic algorithm.

In the following, we introduce each of the mutation methods in detail:

(a) *Coding Confusion*. In the browser parsing process, HTML decoding is usually performed first, and then JavaScript code is parsed. Therefore, attackers can use this feature to encode and obfuscate attack vectors, hide sensitive words in the attack vectors, and bypass the security mechanism. Table 8 shows common types of encoding and their corresponding examples of code-based obfuscation techniques.

(b) *Events Sensitive Words Replacement*. When certain events are detected or filtered, you can continue to ensure the execution of cross-site scripting by replacing other events, such as replacing onclick events with onload events. For specific alternative event types, refer to the event types listed in Table 6.

(c) *Sensitive Functions Replacement*. In some web applications, the alert function may be disabled or filtered because it is always used to test whether there are XSS vulnerabilities, so the interference caused by this strategy can be avoided by replacing other functions. Refer to Table 5 for the replaceable functions in this paper.

(d) *Blank Character Replacement*. In some specific scenarios, we can replace blank characters (e.g., spaces, tabs, etc.) with %0a, %0c, %0d, %00, and so forth; for example, the attack vector < svg onload = alert(1)> after replacement becomes < *svg% 0aonload = alert(1)*>.

(e) *Bracket Replacement*. Replace the parentheses of the function in the JavaScript code with ‘; for example, prompt(1) becomes prompt’1’.

(f) *Attributes and Events Swap Positions*. When there are multiple attributes or events in the attack vector, the detection mechanism in the web application can be bypassed by swapping their positions; for

Table 7: Genetic mutation method (XSS payload bypass method).

Mutation forms	Specific description
Coding confusion	1.HTML encode 2.Unicode encode 3.URL encode 4.Base64
Sensitive words replacement	5.Events sensitive words replacement 6.Sensitive functions replacement 7.Blank character replacement 8.Bracket replacement
Position or form change	9.Attributes and events swap positions 10.Case change 11.Shape transformation of pop-up window function
Add special characters	12.Add a blank character (between the event and the trigger code) 13.Insert the tag into the tag 14.Add notes (between the function and the parentheses) 15.Add some characters before or after the vector

example, it may become < *img onerror = alert(1) src = x*>.

(g) *Case Change*. This mutation method is very common, changing the letter case of an event or function. Some web developers use string matching to find out whether there is an attack on the content submitted by users. Changing the case can often prevent the key characters in the attack vector from being matched.

(h) *Shape Transformation of Pop-Up Window Function*. In many scenarios, some pop-up functions commonly used to test XSS vulnerabilities will be detected, so the function can be deformed to invalidate the detection code. The common way is transforming alert(1) into top['ale'+'rt'](1). By way of splicing to bypass, the detection program cannot find the word "alert."

((i) *Add Blank Characters*. Add 1 3 blank characters between the event of the attack vector and the trigger code, such as space (%20), tab (%09), line feed (%0a), and carriage return (%0d). For example, <img src = *x* onerror%0a%20 = alert(1)>.

(j) *Insert Tag into the Tag*. When the filtering method of the web application is to delete sensitive characters but the recursive method is not used to delete the sensitive characters, the label can be repeatedly inserted, for example,<*scri<script>pt > alert(1) </scri</script>pt*>; if the application deletes "<script>" and "</script>" in the input content, the original attack vector becomes <*script>alert(1) </script*>, and the attack can be successfully implemented.

(k) *Add Notes*. Add the comment symbol/ ∗ ∗ /between the function name of the attack vector and the brackets, and insert several characters randomly between the comment symbols. In this way, the regular expression of the detection program is disturbed, so as to achieve the purpose of bypassing the protection strategy, for example, <svg onload = alert/ ∗ d1dj ∗ /(1)>.

(l) *Add Random Characters before or after the Vector*. Before or after the attack vector, add any combination of characters and numbers (1 5 digits), and the exploit code can be successfully triggered in some specific scenario.

4. Proposed Approach

In this part, we will introduce in detail our proposed method for fuzzing XSS vulnerabilities based on genetic algorithms, including how to generate initial populations and how to define individual gene sequences, pairing methods, fitness calculation, population selection, and mutation. Finally, the fuzzing test of XSS vulnerabilities in web applications is realized, and effective test cases are generated for the test points for vulnerability detection.

4.1. Overview. We proposed a generation model of fuzzy test cases based on the genetic algorithm. The architecture diagram is shown in Figure 2. First of all, we submitted the generated random test data to the web application and then analyzed the response page of the web application and determined the location type of the page output point by judging the different positions in the page where the test data appeared. Then, according to different types of output points and XSS vulnerability grammatical structure to generate the initial individual set corresponding to the type, the data of the initial individual set will not have the attack vector variants introduced in the previous section, only the most primitive individuals that conform to the HTML structure and syntax. Meanwhile, we fuzzed each initial individual on the Web application and calculated the fitness based on the feedback results. After completing the calculation of individual fitness, according to the principles of genetic algorithm, select some individuals for crossover (wherein individuals with high fitness are selected with greater probability). Finally, the individuals who have completed the crossover are mutated. The mutation here is mainly to use the attack vector mutation mentioned in Section 3.4, and randomly select the mutation method to mutate the

Table 8: Coding confusion.

Encoding type	Example
HTML encoding	<svg onload = alert(1)>
Unicode encoding	<svg onload = \ u0061\ u006c\ u0065\ u0072\ u0074(1)>
URL encoding	%3Csvg%20onload%3Dalert(1)%3E
Base64	<iframe src = data:text/html; base64, PHN2ZyBvbmxvYWQ9YWxlcnQoMSk+>

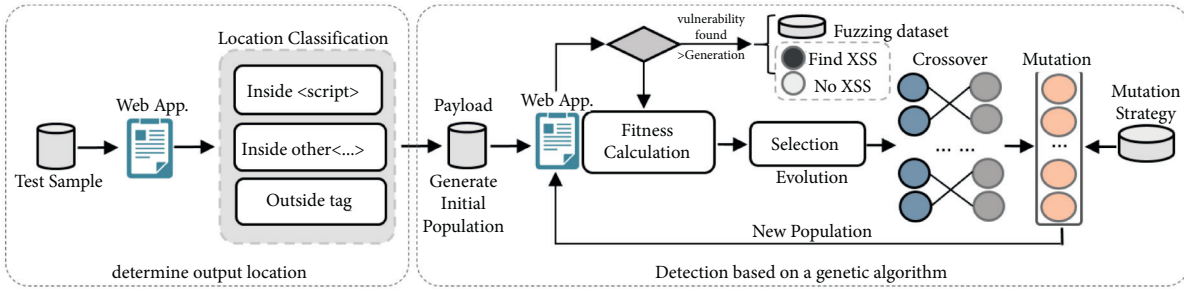

Figure 2: The system architecture of GAXSS.

individuals who have completed the crossover (individuals with high fitness after mutation will be retained). After the mutation is completed, the above process is repeated, and the fuzzing test and fitness calculation are performed again. Until the model finds that the attack vector has been successfully executed or the loop has completed a set number of evolutionary generations, it will end the loop and output the detection results with the final attack vector dataset (dataset can be used for manual test).

4.2. Genetic Algorithm. Genetic algorithm is referred to as GA. Originating from computer simulation research on biological systems, it is a stochastic global search optimization method. The entire algorithm simulates the phenomena of replication, crossover, and mutation that occur in natural selection and genetic behavior [33]. The whole process of genetic algorithm is to start from an initial population, through random selection, crossover, mutation, and so forth, to generate a group of individuals more suitable for the environment, so that the group can evolve to better and better areas in the exploration space. Multiply and evolve, and finally converge to a group of individuals who are most suitable for the environment, so as to obtain the optimal solution or the closest to the optimal solution [34].

Compared with other algorithms, genetic algorithm has outstanding advantages:

(1) The genetic algorithm does not start from a single point but from a group composed of multiple points. The initial dimension is not easily affected by the difference of the starting point.

(2) In the process of exploring the optimal solution, only the fitness information needs to be obtained from the objective function value, and other auxiliary information such as derivatives is not needed. The calculation process is relatively simple.

(3) In the process of exploration, it is not easy to fall into the local optimal solution.

In the genetic algorithm, selection, crossover, and mutation are the three main operation operators of the genetic algorithm. They constitute the genetic operation, giving the genetic algorithm its unique characteristics. The algorithm's expression method is as follows:

$$\mathrm{GA} = (C, E, P_0, M, \phi, \tau, \psi, T). \tag{1}$$

In the expression, C represents the individual coding scheme. In this paper, it represents how to represent the cross-site scripting load as a DNA code string; E represents the individual fitness evaluation function; P_0 represents the initial population; M represents the population size; ϕ represents the selection calculation Sub; τ represents the crossover operator; ψ represents the mutation operator; T represents the algorithm termination condition; in this paper, it stops when the genetic algorithm has evolved for 30 generations.

4.2.1. Individual Code. The mapping from the solution of the problem to the genotype is called encoding, which is a conversion method that converts the feasible solution of a problem from its solution space to the search space of the genetic algorithm [35]. Before the genetic algorithm is executed, the data is expressed as the genotype string structure data of the genetic algorithm (considered as a chromosome). In the paper, the XSS payload is converted into this structured data. Common encoding forms are binary encoding, gray code encoding, floating-point number encoding, and so forth. We choose multiple types of digital encoding.

We designed the DNA structure converted from the XSS payload into three parts: closing (expressed by C), main (expressed by B), and mutation (expressed by M). The

individual DNA structure can be expressed as shown in Figure 3:

(1) *Closing Part*. This part consists of 4 digits from 0 to 4, each of which represents a closed character, as shown in Table 9.

(2) *Main Part*. The main part consists of 6 numbers, which, respectively, represent the label ($B1$), event ($B2$), XSS exploit code ($B3$), attribute ($B4$), pseudoprotocol ($B5$), and backward closing character ($B6$). The backward closing character is only used when the input point is between the <script> tags. It is used to close the JavaScript syntax, such as//, which is used to comment out the following code. The numbers B1–B6 of the main body part can be 0, which means that the corresponding related parts are not selected (refer to Section 3.2).

(3) *Mutation Part.* This part is represented by N numbers ($0 \le N \le m$, where m represents the number of all mutation types), where N represents how many types of mutations exist. When $N = 0$, it means that there is no mutation in this individual DNA, and the mutation number is shown in Table 7 in Section 3.4.

When the genetic algorithm needs to decode the gene sequence into the original structure of the payload data or the XSS payload needs to be encoded into the DNA sequence, use the comparison table of each part for encoding or decoding, and the structure is as illustrated in Figure 3. For example, the individual DNA sequence is "4 2 0 4 3 1 3 2 0 0 10 15," which can be decoded as '><img dynsrc = x oNerRor = prOmPt(1)>ecodi; and <script>alert(1)</script> can be encoded as "4 4 4 4 5 0 1 0 0 0."

4.2.2. Fitness. In the genetic algorithm, the fitness function indicates the pros and cons of the individual or the solution. For different problems, the fitness function is defined in different ways [33]. In the algorithm, fitness is used to select individuals who are more suitable for web fuzz testing. Through our research on XSS vulnerabilities, we analyzed the possible responses in the fuzzing test and obtained four factors that have an impact on the fitness function: whether the code can be executed successfully, whether the closure is completed, the similarity of input and output, and the filtered situation:

(1) Examples of whether the code can be executed successfully are a successful pop-up window, a successful jump, and so forth. This indicates that the test individual fully complies with the XSS exploit method and grammatical structure, so when the exploit code is executed successfully, we set $\mathrm{Ex}(I, O) = 2$; otherwise, $\mathrm{Ex}(I, O) = 0$.

(2) Successful completion of the closure is the key to the exploitation of XSS vulnerabilities and the key to the successful introduction of exploit code. If the previous piece of code cannot be successfully closed and it conforms to the grammar, it is very likely that the input content will be treated as a string and will not be parsed by the browser. By obtaining the HTML source code of the output point context of the page response, we can calculate that the total number of symbols that need to be closed for the output point is $c1$, and the number of unclosed symbols is set to $c2$. So the fitness of the closed part is as follows: $\mathrm{CLOSED}(I, O) = (c1 - c2)/(c1 + 1e3)$. Adding a decimal to the denominator is to avoid the situation that the denominator becomes 0 when the denominator does not need to be closed in a special case.

$$\underbrace{C_1C_2C_3C_4}_{\text{Closing part}}\ \underbrace{B_1B_2B_3B_4B_5B_6}_{\text{Main part}}\ \underbrace{M_1M_4}_{\text{Mutation part}}$$

Figure 3: Individual DNA chain structure (from XSS payload).

(3) The input and output similarity is the result of comparing the input test vector $s(I)$ with the output vector (web response content) $s(O)$ processed by the server. The similarity is calculated using Levenshtein distance [36], which is denoted as Ldis. Levenshtein distance describes the minimum number of operations required to transform one string into another. In general, the higher the similarity, the less the filtered part of the fuzzy test vector. What we need to explain here is that when the input vector is a vector after coding mutation (such as items 1 to 4 in Table 7), in the distance calculation, the input data $s(I)$ is the vector before coding mutation. The similarity of input and output is expressed as $\mathrm{Dis}(I, O) = 1/(1 + \mathrm{Ldis}(s(I), s(O)))$; the higher $\mathrm{Dis}(I, O)$ score indicates that the input and output contents are more similar.

(4) In fuzzing individuals, the closed part and the JavaScript part of the exploit code are indispensable. Therefore, when this part of the code becomes invalid due to being encoded and filtered, the fitness will be reduced. We set $Pu1 = -0.2$ when the closed part is invalid and $Pu2 = -0.5$ when JavaScript is filtered; in other cases, $Pu1$ and $Pu2$ are 0 by default. So we set the score of this part to $Pu(I, O)$, where $Pu(I, O) = Pu1 + Pu2$.

When calculating the fitness of an individual, combining the effects of the above factors on the fitness, the fitness of the genetic algorithm for the web fuzzing test is as follows:

$$F(I, O) = \mathrm{Ex}(I, O) + \mathrm{CLOSED}(I, O) + \mathrm{Dis}(I, O) + Pu(I, O). \tag{2}$$

4.2.3. Crossover and Mutation. Before the cross-mutation process, first, calculate the fitness value corresponding to each individual, and then select half of the test individual for cross-mutation. The selected individual set is set as pop_a, and the total set of individuals is pop. We set the probability of each individual being selected as P, and one of the points is P_i. Individuals with higher fitness are easier to be selected, as shown in the following equation:

Table 9: Character description of the closed part.

Number	Character
0	>
1	)
2	’
3	”
4	N/A

$$p_i = \frac{F_i(I,O)}{\sum_{n=1}^{M} F_n(I,O)}. \quad (3)$$

Then, select each individual in pop_a and a random individual in pop for crossover and mutation. Each pair of individuals will produce two offspring. The crossover process is shown in Figure 4.

As seen in Figure 4, we set the total number of individual sets (pop) to be m; then pop_a is a subset of pop with a number of n ($n = m/2$). Each item in pop_a is randomly crossed with an individual in the pop set to generate a new subset, and m offspring (e.g., $a1 + p2 - > a1p2\_1, a1p2\_2$) can be obtained. Merge the new offspring set with the original individual set pop and sort according to the fitness value from large to small. Ultimately, we selected top m individuals as the next-generation pop set according to their fitness. The advantage of this advanced method is that the generated offspring do not directly replace the parent but choose whether to be retained according to their fitness. When the parent and the offspring have high fitness, their individuals will be retained, and other individuals with low fitness will be eliminated.

In the genetic algorithm, to ensure the randomness of the species, we will set a probability value P_m (set to 0.8 in this paper); that is, the probability of P_m crosses normally, and the probability of $1 - P_m$ makes the offspring directly equal to the parent in pop_a (no crossover). During the crossover process, the pair will generate a random Boolean sequence based on the length of the longer sequence number and then perform individual crossover to generate a new individual, as shown in Figure 5.

The mutation process is carried out after the crossover. Just like gene mutation, the offspring of each individual after the crossover has a certain probability of mutation. Each of the attack vector mutation methods introduced in Section 3.4 has a certain probability to occur on new individuals, and we set the probability to 0.1, which means that the probability of each mutation is 10%. This kind of mutation is different from the bit mutation in the traditional genetic algorithm. If some kind of mutation occurs in this paper, the serial number corresponding to the mutation method is added to the back of the individual serial number; for example, “4 4 4 4 5 0 1 0 0 0″ becomes “4 4 4 4 5 0 1 0 0 0 10″ after mutation. Through our testing and analysis, we found that the mutated individuals will more likely bypass those web applications that have security filtering mechanisms and execute the exploit code.

To prevent the program from falling into an infinite loop, we set the iteration limit D for the vulnerability detection of each test point (set the number of iterations $D = 30$). After 30 generations of evolution, if the vulnerability cannot be found, it is considered that there may be no vulnerability in the location, the program will be stopped, and output the algorithm judgment result and the final set of individuals.

5. Experiment

Based on the prototype system, we evaluated the parameter selection of the GAXSS system. In addition, in the experiments, the GAXSS system was compared with other black-box XSS scanners and detection methods to evaluate the detection capability and efficiency of the system. The experimental results show that the overall detection effect of the GAXSS is relatively excellent, and applications that have not detected XSS vulnerabilities can be manually tested through the output individual set, which is better than other detection systems.

5.1. Environment. The experimental environment we designed in this paper was all run under the MacOS 10.15.7 operating system, which is equipped with a 2.6 GHz 6-core Intel Core-i7 processor and 16 G RAM DDR4 at 2667 MHz. All the functional modules in the paper were developed and run under the Python 3.6 environment.

To evaluate the effectiveness of the detection model proposed in the paper, we selected 6 different web applications with different complexity to conduct experiments. As shown in Table 10, GAXSS can detect at least one real XSS vulnerability in all applications. Moreover, to compare the efficiency of the detection methods, we also selected 4 well-known black-box XSS scanners and three advanced detection schemes proposed by other researchers in recent years for comparison with GAXSS. To be clear, in the experiment, we tested the input points that have been prepared by the crawler program. If an input point was found to be vulnerable, this vulnerability would not be considered whether the type and form are the same as other vulnerabilities (a page may have multiple output points). Whether the input points collected by the crawler program are comprehensive is beyond the scope of this paper.

5.2. Evaluation. In a binary decision problem, the result of each classification may be positive or negative, so the decision result can be represented by a confusion matrix structure. In this paper, the confusion matrix is divided into 4 categories: T_p (True Positive) represents the number of samples that have correctly identified XSS vulnerabilities, and F_p (False Positive) represents the number of samples that have no vulnerabilities identified as XSS vulnerabilities. Similarly, T_n (True Negative) indicates that samples without vulnerabilities are correctly classified as nonvulnerable samples, and F_n (False Negative) indicates the number of samples that have vulnerabilities but have not been detected. Based on the confusion matrix, we used accuracy, precision, and recall to evaluate the experimental results. The formulas are expressed as follows:

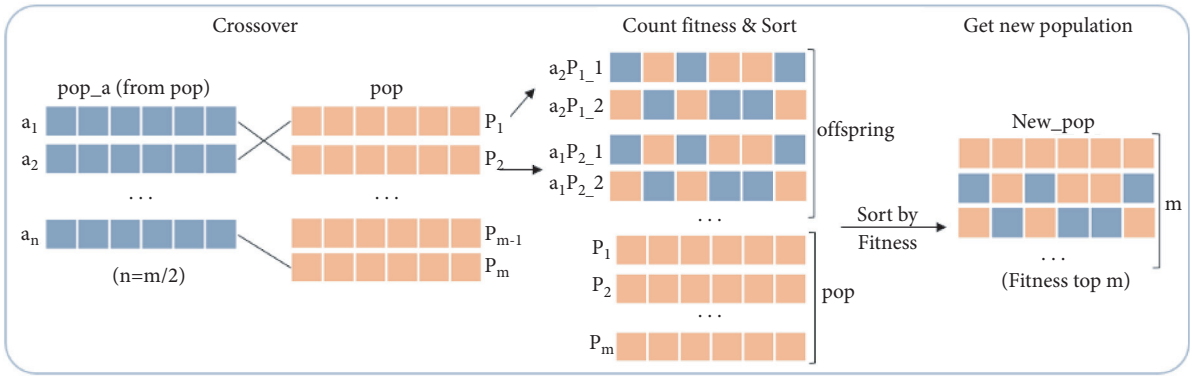

Figure 4: Example of crossover and generating new population processes.

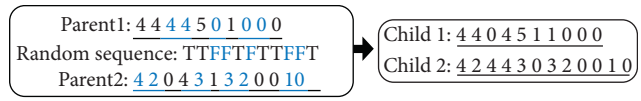

Figure 5: Crossover example.

Table 10: Tested web applications.

Application	Description	Version
DedeCMS	Web CMS	5.7
WebGoat	Intentionally vulnerable	5.4
WordPress	Blog CMS	3.2.1
EmpireCMS	Web CMS	7.5
phpBB	Forum	2.0
Self-built website	For XSS test	2021_07

$$\text{accuracy} = \frac{T_p + T_n}{\left(T_p + F_n\right) + \left(F_p + T_n\right)},$$
$$\text{precision} = \frac{T_p}{T_p + F_p}, \tag{4}$$
$$\text{recall} = \frac{T_p}{T_p + F_n}.$$

5.3. Results and Analysis. In order to verify the effectiveness and advancement of the XSS fuzzy test detection model based on the genetic algorithm, we designed three experiments to prove it. The first experiment is to conduct experiments on the parameters of the model and compare the detection effects of XSS vulnerabilities under different parameter environments. Through analysis, we conducted experiments on the three main parameters in the genetic algorithm: the ratio of crossover, the probability of mutation, and the selected evolutionary algebra. The second experiment is to verify the effectiveness of the fuzzing method. By using GAXSS to test different web applications with known vulnerabilities, it verifies the ability of the method proposed in the paper. Finally, we used multiple XSS scanners and advanced detection schemes to compare the scanning results and detection times of different locations and demonstrated the good performance of GAXSS in vulnerability detection.

5.3.1. Parameter Selection Experiment. In the fuzzing test of XSS vulnerabilities, it is very important to choose a suitable algorithm to solve the problem. Different parameters in the algorithm will also greatly affect the effect and performance of the model.

We selected the locally built EmpireCMS v7.5 as the experimental object, used different parameters for the same designated input point to conduct a control experiment, and selected relatively stable experimental results as the basis for parameter selection. Because we selected the same input point, the initial individual set was the same. In the crossover process, 0.3 times, 0.5 times, and 0.7 times the total number of individuals N were selected for crossover, the mutation probability was 0.1, and then 30 generations of evolution checked the average fitness of the 10 individuals with the highest fitness in the individual set. Due to the strong randomness of this method, the data of each experiment cannot be the same, but the overall trend is consistent. The experimental results are shown in Figure 6(a). In addition, in the mutation process, by mutating the XSS payload, the success rate of vulnerability exploitation could be increased, but too high or too low mutation probability would also affect the final result. We took the mutation probability p as

0.2, 0.1, and 0.05 and the crossover ratio as 0.5. The experimental data is shown in Figure 6(b).

It is not difficult to see from Figure 6(a) that different ratios used for crossover have little effect on the overall results. The results of the three parameter choices can make the fitness more than 2.0, and the performance of $pop_a = N * 0.5$ is slightly better than others. An inappropriate crossover ratio may reduce the evolution gradient, so, in subsequent experiments, we chose the crossover ratio to be 0.5. In addition, we can find from Figure 6(b) that when the mutation probability is 0.2, it will seriously affect the genetic algorithm's optimal selection of individuals, and when the mutation probability is 0.1 and 0.05, the performance is better. Our analysis found that when the individual is close to the executable payload, the higher mutation rate may make the individual mutate and become unsuitable. Therefore, in subsequent experiments, we choose the mutation probability to be 0.1.

The termination condition of the entire algorithm is to terminate when the set number of evolutions is completed. Too few times will make the individual set not reach a better set state, and too many times will waste computing resources, and it is also easy to make the set fall into a local optimum solution. We designed experiments to test the input points of three different applications with the same parameters and observe the changes in fitness. The experimental results are shown in Figure 7. We can observe the test results for the three applications. The fitness average tends to stabilize around the 30th generation, so we set the model to end the computation after looping 30 times.

5.3.2. Method Validity Experiment. In the second experiment, we designed to use the GAXSS model to test for multiple different input points of 6 applications. In the preparation phase, we used the crawler to obtain potential test points that may have vulnerabilities in each application and manually verified the vulnerability of the test points by researchers. Moreover, it should be noted that, to save time, we have set that as long as there is an individual that can trigger the vulnerability and execute successfully in the generated individual set, the loop of the program will be terminated, and there is no need to wait for the evolution to complete 30 generations. We used the method proposed in this paper to detect the collected interaction points and proved that the method is effective through the precision rate, accuracy rate, and recall rate of the experiment. The result data of the experiment is shown in Table 11.

It can be seen from Table 11 that the method proposed in this paper can detect and discover vulnerabilities in web applications. Our research found that because judging whether a vulnerability is found is verified by judging whether it can generate executable exploit code, it is basically impossible to execute the exploit code at a test point without vulnerabilities, so there will be no False Positive in the experiment but only cases of underreporting. In the detection of all samples, a total of 27 vulnerabilities existed, and 22 real vulnerabilities were detected by the program with a precision rate of 100%, an accuracy rate of 97.5%, and a recall rate of 81.5%. The running time of the detection program is proportional to the number of points to be detected, and the total detection time is meaningless, so we count the average time to detect vulnerabilities in the table. The average time of WordPress and Self-built applications is slightly longer, because these two applications have certain security mechanisms, and the payload individuals to be constructed are more complex than others, so more evolution and mutation are required to find suitable individuals to trigger the vulnerability. The experimental data proved that the method is effective for XSS vulnerability detection.

We analyzed the undetected vulnerabilities and found that these vulnerabilities are exploited in a very specific form, which leads to model detection failure. In some cases, it is necessary to construct very special code to bypass the security detection of the application, or the security policy setting is more complicated, and the test functions used in the detection model are all filtered.

5.3.3. Scanner Comparison. In the third experiment, we used different scanning tools to test the samples collected by the crawler program in the second experiment and compared the differences between our proposed method and classic open-source tools and other detection schemes. In the experiment, we used the 4 tools of Wapiti, w3af, XSSer, and XSStrike and the three detection schemes of XSS-Unit testing [29], TT-XSS [30], and WVF [37] proposed by other researchers in recent years and GAXSS test results for comparison. The GAXSS parameters were set as follows: mutation probability of 0.1 and crossover ratio of 0.5. The program will end the loop when an executable individual is found; otherwise, it will stop until 30 evolutions are completed and the final generation of individual sets is output. After the scanning is completed, we can also perform manual testing by outputting high fitness individuals in the results. We detected 202 test points in the second experiment; 27 points were vulnerable and 175 points were not vulnerable. The results of the experiment are shown in Table 12.

As shown in Table 12, since TT-XSS [30] can only detect DOM-type vulnerabilities, the number of vulnerabilities found is small, and the number detected by other detection schemes is not much different. GAXSS is superior to other scanning systems in terms of detection accuracy and recall rate, and the accuracy rate exceeds 97%, indicating that our proposed method has good performance and is higher than the average level in XSS detection. The False Positives of all scanners are 0. However, the ability to detect complex vulnerabilities needs to be improved. In addition, unlike other scanners, the output of GAXSS can be not only the conclusion of whether there are vulnerabilities but also the set of evolved individuals. Even if the program does not find the vulnerability of the web application, it can also use the highly adaptive individual set to test the application again by manual means. Because highly adaptive individuals are closer to the reasonable way of exploiting vulnerabilities, the efficiency of manual detection can be greatly improved.

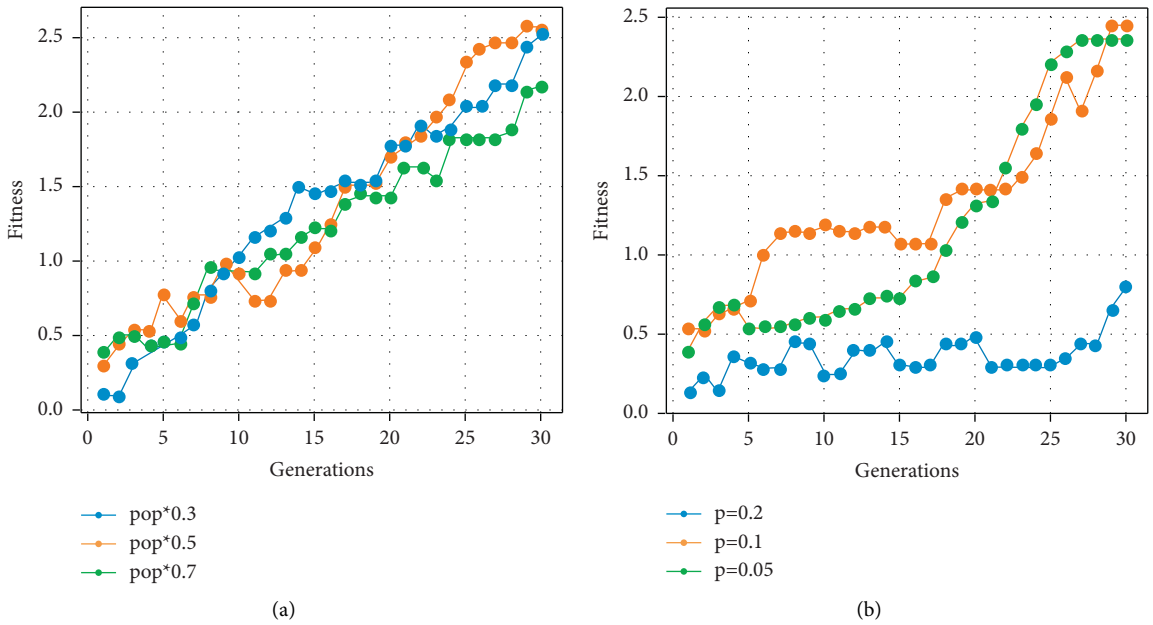

Figure 6: (a) Crossover parameter selection. (b) Mutation probability selection.

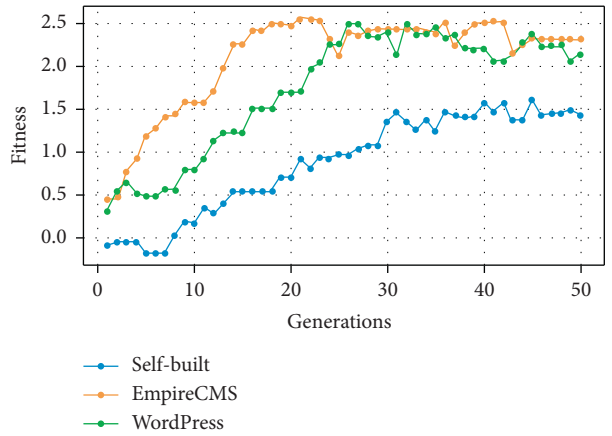

Figure 7: Generation selection.

Table 11: Detection capabilities for different applications.

Application	Test points	XSS actually exists	True XSS found	Accuracy	Recall	False Positive	Average time/vul. (s)
DedeCMS	32	4	4	1.0	1.0	0	54
WebGoat	52	6	5	0.981	0.833	0	46
WordPress	28	5	4	0.964	0.8	0	79
EmpireCMS	30	3	2	0.967	0.667	0	24
phpBB	42	1	1	1.0	1.0	0	36
Self-built web	18	8	6	0.889	0.75	0	72
Total	202	27	22	0.975	0.815	0	-

In terms of time overhead, we used the above 8 detection methods to detect one of the vulnerabilities in the 6 applications in the first experiment multiple times (5 times) and recorded the running time of these 30 rounds of experiments to observe the difference in efficiency between various schemes. The distribution of 30 test times for each scheme is represented by a box plot, as shown in Figure 8.

Due to the randomness of the genetic algorithm characteristics of the GAXSS model, the time consumption of detecting vulnerabilities has a large numerical span, while other tools and solutions have fixed processes, and the time consumption of detection is relatively stable. The detection method of GAXSS can be quickly optimized to find individuals that can trigger vulnerabilities, and it does not need

Table 12: Scanners detection performance (27 vul. in total).

Scanner	XSS found	Accuracy	Recall	False Positive	Manual test
Wapiti	18/27	0.955	0.667	0	F
w3af	20/27	0.965	0.741	0	F
XSSer	16/27	0.946	0.593	0	F
XSStrike	17/27	0.950	0.630	0	F
XSS-unit	17/27	0.950	0.630	0	F
TT-XSS	10/27	0.916	0.370	0	F
WVF	18/27	0.955	0.667	0	F
GAXSS	22/27	0.975	0.815	0	T

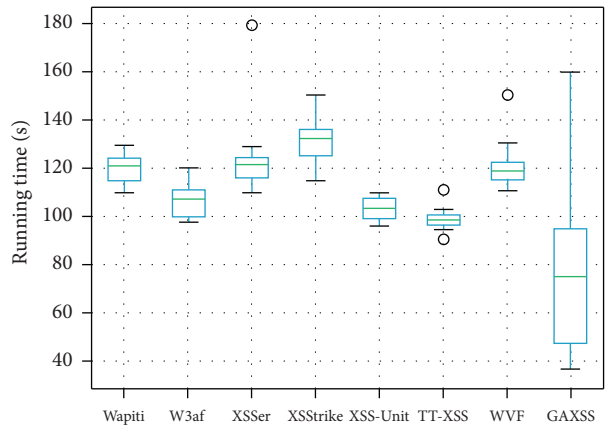

Figure 8: Vulnerability detection time cost box plot.

to run all processes every time, so the median overall time consumption is the smallest among these solutions, only 75.5 s, the minimum time is 36 s, and the maximum is 160 s.

6. Conclusion

This paper proposed a cross-site scripting vulnerability fuzzing test model based on a genetic algorithm. This model can dynamically generate test individuals for different types of input points and then use the optimized genetic algorithm to select and mutate individuals. Finally, through finding executable payload to discover applications vulnerabilities, in the wild, GAXSS can be used to detect possible vulnerable interactive functions in web applications, and developers can fix the vulnerabilities in time to avoid more terrible losses. We first gave an overview of the relevant principles and research motivations of XSS vulnerabilities and then analyzed the XSS structure and bypass detection methods. Then, a vector generation detection model based on a genetic algorithm is proposed to detect XSS vulnerabilities. Finally, through three groups of experiments, it is proved that the parameter selection of the model itself is reasonable, and it has excellent detection performance of XSS vulnerabilities. Particularly, our proposed method can manually test the undetected vulnerability points through the evolved individual set and use high fitness individuals to improve the efficiency of manual detection. Other scanners do not have this kind of function. In future work, parallel detection performance can also be optimized to improve batch detection efficiency.

Data Availability

In order to verify the validity of the model, we used actual site-building data in the experiment, which could be downloaded from the Internet and built locally. The data download address is as follows: DedeCMS: https://www.dedecms.com/; WebGoat: https://github.com/WebGoat/WebGoat/releases; WordPress: https://github.com/WordPress/WordPress; EmpireCMS: https://www.phome.net/; phpBB: https://github.com/phpbb/phpbb-app; Self-built website: the self-built website used to test the effectiveness of vulnerability detection is not open source.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (U20B2045).

References

[1] CNNIC, "The 47th statistical report on Internet development in China," 2021, http://www.cnnic.net.cn/hlwfzyj/hlwxzbg/hlwtjbg/202102/P020210203334633480104.pdf.

[2] OWASP, "OWASP top 10 - 2017," 2021, https://owasp.org/www-pdf-archive/OWASP_Top_10-2017_(en).pdf.pdf.

[3] L. Constantin, *New Chinese Social Networking Worm Discoverd*, 2021, http://news.softpedia.com/news/New-Chinese-Social-Networking-Worm-Discovered-120021.shtml.

[4] Samy, "Technical explanation of the MySpace worm," 2021, https://samy.pl/popluar/tech.html.

[5] G. Cluley, *Cross-platform Boonana Trojan Targets Facebook Users*, 2021, https://nakedsecurity.sophos.com/2010/10/28/cross-platform-worm-targets-facebook-users/.

[6] Hackagon, "XSS attack," 2016, http://hackagon.com/xss-attack/.

[7] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[8] Y. Peng, *Research and System Implementation of Key Technologies of Web Vulnerability Detection*, Beijing University of Posts and Telecommunications, Beijing, China, 2014.

[9] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, no. 1, pp. 512–530, 2017.

[10] OWASP, "XSS filter evasion cheat sheet," 2021, https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet.

[11] D. S. Fowler, J. Bryans, and M. Cheah, "A method for constructing automotive cybersecurity tests, a CAN fuzz testing example," in *Proceedings of the 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion*, pp. 1–8, Sofia, Bulgaria, July 2019.

[12] J. Liao, T. Tsai, C. He, and C. Tien, "SoliAudit: smart contract vulnerability assessment based on machine learning and fuzz testing," in *Proceedings of the 2019 Sixth International Conference on Internet of Things: Systems, Management and Security*, pp. 458–465, Granada, Spain, October 2019.

[13] H. Choi, S. Hong, S. Cho, and Y. Kim, "HXD: Hybrid XSS detection by using a headless browser," in *Proceedings of the 2017 4th International Conference on Computer Applications and Information Processing Technology*, pp. 1–4, Kuta Bali, Indonesia, August 2017.

[14] H. Shahriar and M. Zulkernine, "Mutec:Mutation-basedtestingofcross site scripting," in *Proceedings of the ICSE Workshop on Software Engineering for Secure Systems*, pp. 47–53, Vancouver, BC, Canada, May 2009.

[15] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pp. 6–263, IEEE, Berkeley/Oakland, CA, USA, May 2006.

[16] F. Duchene, R. Groz, and S. Rawat, "XSS vulnerability detection using model inference assisted evolutionary fuzzing," in *Proceedings of the IEEE 5th International Conference on Software Testing,Verification and Validation*, pp. 815–817, Montreal, QC, Canada, April 2012.

[17] Z. Tang, H. Zhu, Z. Cao, and S. Zhao, "L-wmxd: lexical based webmail XSS discoverer," in *Proceedings of the 2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 976–981, IEEE, Shanghai, China, April 2011.

[18] Y. Zhang, X. Wang, P. Wang, and L. Liu, "Detecting cross site scripting vulnerabilities introduced by HTML5," in *Proceedings of the 2014 11th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pp. 319–323, Chon Buri, Thailand, May 2014.

[19] X. Hou, X. Zhao, M. Wu, R. Ma, and Y.-P. Chen, "A dynamic detection technique for XSS vulnerabilities," in *Proceedings of the 2018 4th Annual International Conference on Network and Information Systems for Computers (ICNISC)*, pp. 34–43, IEEE, Wuhan, China, Apri 2018.

[20] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: a client-side solution for mitigating cross-site scripting attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 330–337, ACM, Dijon, France, April 2006.

[21] A. Barth, C. Jackson, and J. Mitchell, "Robust defenses for cross- site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*, pp. 75–88, ACM, Virginia, Alexandria, USA, October 2008.

[22] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: server-side detec- tion of cross-site scripting attacks," in *Proceedings of the 2008 Annual Computer Security Applications Conference*, pp. 335–344, IEEE, Anaheim, CA, USA, December 2008.

[23] P. Bisht and V. Venkatakrishnan, "XSS-GUARD: precise dynamic prevention of cross-site scripting attacks," *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 5137, pp. 23–43, Springer, Berlin, Heidelberg, 2008.

[24] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 2008 ACM/IEEE 30th International Conference on Software Engineering*, pp. 171–180, IEEE, Leipzig, Germany, May 2009.

[25] Z. Liu, Y. Fang, C. Huang, and J. Han, "GraphXSS: an efficient XSS payload detection approach based on graph convolutional network," *Computers & Security*, vol. 114, p. 102597, 2022.

[26] F. Duchene, S. Rawat, and J. L. Richier, "KameleonFuzz: evolutionary fuzzing for black-box XSS detection," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pp. 37–48, San Antonio, TX, USA, March 2014.

[27] J. Bozic, B. Garn, and I. Kapsalis, "Attack pattern-based combinatorial testing with constraints for Web security testing," in *Proceedings of the 2015 IEEE International Conference on Software Quality,Reliability and Security*, pp. 207–212, Vancouver, BC, Canada, August 2015.

[28] F. Duchene, S. Rawat, and J. L. Richier, "KameleonFuzz: evolutionary fuzzing for black-box XSS detection," in *Proceedings of the 4th ACM conference on Data and application security and privacy*, pp. 37–48, Texas, San Antonio, USA, March 2014.

[29] M. Mohammadi, B. Chu, and H. R. Lipford, "Detecting cross-site scripting vulnerabilities through automated unit testing," in *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 364–373, Prague, Czech Republic, July 2017.

[30] R. Wang, G. Xu, X. Zeng, X. Li, and Z. Feng, "TT-XSS: a novel taint tracking based dynamic detection framework for DOM Cross-Site Scripting," *Journal of Parallel and Distributed Computing*, vol. 118, pp. 100–106, 2018.

[31] C. Chen and Y. H. Zhou, "XSS vulnerability mining based on fuzzing test and genetic algorithm," *Computer Science*, vol. 43, no. S1, pp. 328–331, 2016.

[32] G. Goswami, H. Nazrul, K. B. Dhruba, and K. Jugal, "An unsupervised method for detection of XSS attack," *International Journal on Network Security*, vol. 19, pp. 761–775, 2017.

[33] S. Mirjalili, *Genetic Algorithm. Evolutionary Algorithms and Neural Networks*, pp. 43–55, Springer, Cham, Switzerland, 2019.

[34] X. Bian and L. Mi, "Research progress of genetic algorithm theory and its application," *Application Research of Computers*, vol. 27, no. 7, pp. 2425–2429, 2010.

[35] I. Hydara and H. Zulzalil, "An approach for cross-site scripting detection and removal based on genetic algorithms,"

in *Proceedings of the 9th International Conference on Software Engineering Advances*, pp. 227–232, Nice, France, October 2014.

[36] L. Yujian and L. Bo, "A normalized Levenshtein distance metric," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[37] M. N. khalid, M. Iqbal, K. Rasheed, and M. M. Abid, "Web vulnerability finder (WVF): automated black- box web vulnerability scanner," *International Journal of Information Technology and Computer Science*, vol. 12, no. 4, pp. 38–46, 2020.